GT-EP: A HIGH PERFORMANCE REAL-TIME PROCESSOR

SPECIAL TECHNICAL REPORT

REPORT NO. STR-0142-90-0011

September 12, 1990

GUIDANCE, NAVIGATION AND CONTROL
DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Contract Data Requirements List Item A004

Period Covered: Not Applicable

Type Report: As Required

GT-EP: A HIGH PERFORMANCE REAL-TIME PROCESSOR

September 12, 1990

Authors

W. S. Tan, C. O. Alford and Sam H. Russ

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Eugene L. Sanders
USASDC
Contract Monitor

Cecil O. Alford
Georgia Tech
Project Director

Computer Engineering
Research Laboratory

Georgia Tech

GT–EP : A High Performance Real–Time Processor

Dr. W. S. Tan, Dr. C. O. Alford, and S. H. Russ
Computer Engineering Research Laboratory
400 Tenth Street
Atlanta, GA 30332–0540
Tel. (404) 894–2533
Fax. (404) 894–3120

Special Technical Report CERL008–0170.1, August 1990

TABLE OF CONTENTS

GEORGIA TECH GT–EP
HIGH PERFORMANCE REAL–TIME PROCESSOR

1. Background Information

The GT–EP is the result of a research effort at Georgia Tech to overcome several problems associated with existing conventional complex instruction set computer (CISC) and reduced instruction set computer (RISC) architectures.

Most of these CISC and RISC architectures are derivations of the von Neumann architecture model [Patt82, Gupt83]. Although the technologies that produced the von Neumann computer architecture have changed dramatically, the notion of today "computer" is still identified with the concept of the 1940's. The two major parts of a von Neumann computer are the central processing unit and the central memory unit. Operations are carried out by moving data back and forth between these two central units. With the advancement of device technology, the link between the central processing unit and the central memory unit has become a performance bottleneck (identified as the *von Neumann bottleneck* in [Back78]).

Secondly, because of the increasing complexity of today modern applications, most software is written in high level languages. Significant amount of performance can be potentially lost in the compilation process from a high level language to a target machine code. This loss of performance is referred as the *semantic gap* and is due to a mismatch between the expression of software algorithms and the underlying hardware processor architecture that executes them [Myer78].

The *von Neumann bottleneck* and the *semantic gap* account for a significantly low utilization of the processing capabilities of conventional computer architectures. Quite often, peak performance is quoted for a particular processor, whereas in reality 10% of the peak is difficult to attain for typical applications.

The GT–EP processor is designed to overcome the von Neumann bottleneck and to close the architectural semantic gap that exists in today generation of CISC and RISC architectures. The architecture was systematically derived from a set of optimality criteria extracted from language constructs commonly used in many high level languages. The result is an architecture that yields high processor utilization across a wide spectrum of applications.

2. Architectural Highlights

The primary functional modules of the GT–EP processor are shown in Figure 1. The Instruction Address Generation (IAG) Module is responsible for calculating the addresses for instruction fetches, specifying ALU operations, and directing I/O traffic. The Data Address Generation (DAG) Module calculates two addresss fields for data fetches and one address field for data stores.

The GT–EP handles I/O data significantly different from conventional processor bus access. All input data flows into the GT–EP processor through the ALU and all output data from the GT–EP processor flows out of the ALU. An analogy of the way the GT–EP processor handles I/O data can be made to that

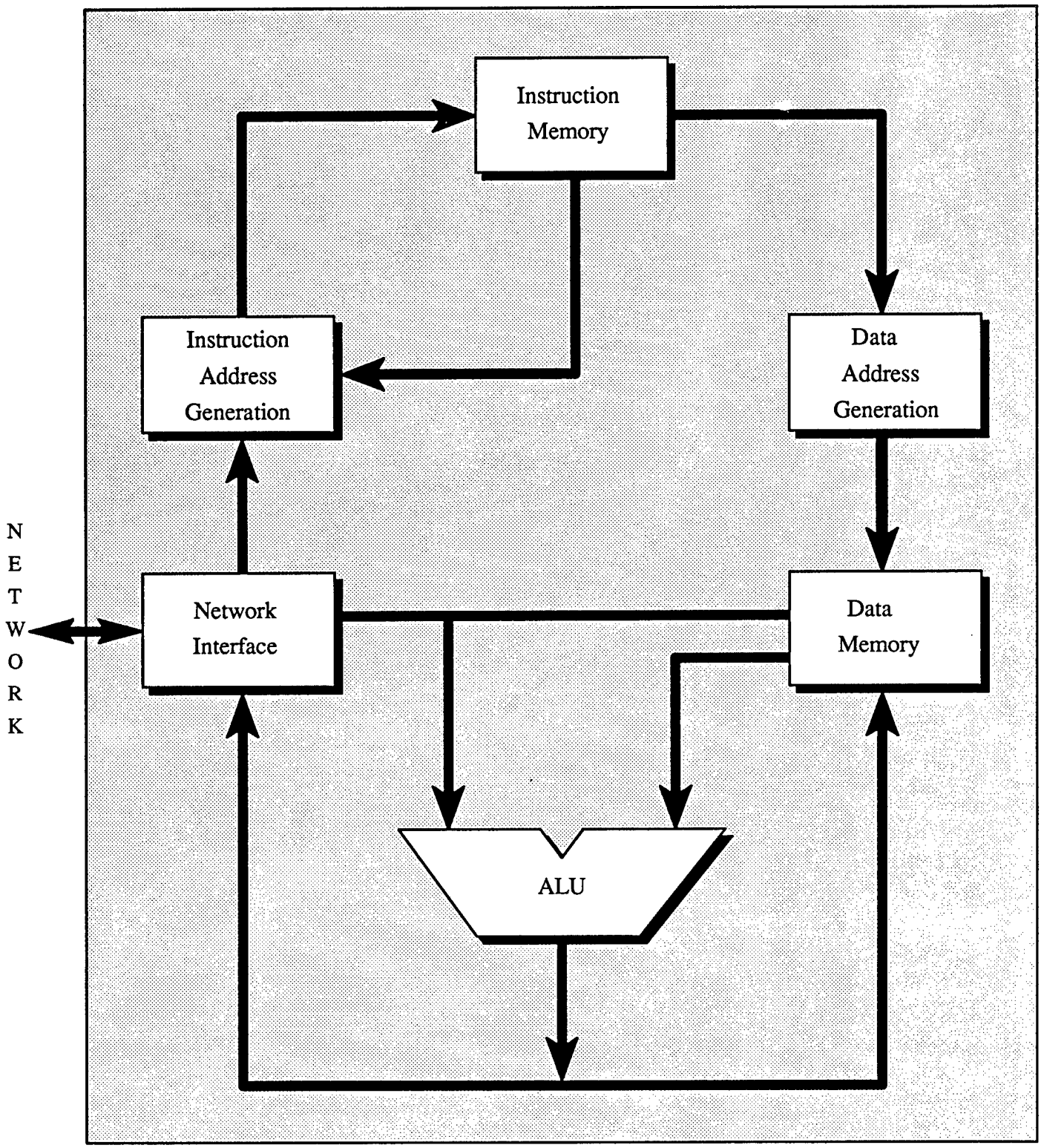

Figure 1. GT–EP Functional Modules

of a dataflow architecture. In the GT–EP processor, the ALU is the central core (data node). Data (tokens) flows into the ALU, is operated on, and produces new data (new token). Up to 16 peripheral devices (arrows directed to the node) can be supported by the GT–EP. The Data Memory and the Network Interface Modules depicted in Figure 1 are two such peripheral devices. This aspect of the architecutre makes the GT–EP very efficient in handling numerically intensive, and I/O intensive problems.

The GT–EP supports 16 flexible peripheral devices. The 16 I/O peripheral devices are divided into 4 fast synchronous (FS) devices, 8 slow synchronous (SS) devices, and 4 slow asynchronous (SS) devices. The FS devices drive separate device status signals into the GT–EP for maximum input and output operations. The SS devices use a set of generic handshake signals to interface with the GT–EP. The SA devices allow the GT–EP to interface with peripheral devices that operate from a different clocking scheme which differs from the GT–EP.

The GT–EP provides hardware support to process instructions at the kernel and user levels. Several instructions are limited to execution at the kernel level. An attempt to execute privileged kernel instructions at the user level will cause a hardware trap. Multi–tasking is inherently supported at the hardware level. The user level program space is relocatable at run–time.

Hardware implementation of an automatic operand dependency check, allows the GT–EP to process instructions as though the processor is operating in a flow through, non pipelining fashion. Branch look-ahead circuitry allows a conditional branch to proceed either assumming the branch will or will not take place. This feature significantly reduces compiler complexity and avoids unnecessary *nops* associated with pipeline architectures.

Interactions with external devices can be effectively handled by the GT–EP processor through interrupts. The GT–EP has a very low interrupt latency. It responds to an external interrupt request in a maximum of 5 cycles. Its I/O dataflow structure significantly reduces the software overhead required for context switching. Nine external interrupt sources can be directly connected to the GT–EP processor.

The GT–EP can also interface efficiently with external I/O devices on a polling basis. Three external signals can be connected to the GT–EP processor to poll the status of external peripheral devices. Based on these signals, the GT–EP can perform a single cycle conditional branch instruction.

Two, 26–bit counters are available for general usage. These timers can be used to schedule time slicing between tasks and to execute tasks that require periodic services. A third timer is a dedicated monitoring timer to prevent the processor from being hung up in a freeze condition while accessing peripheral devices and intruction memory that do not respond appropriately to an I/O or intruction fetch request.

The GT–EP is designed with built–in testability circuitry. In operation, the processor can be stopped and test instructions can be inserted into its instruction stream. The GT–EP internal states can then be examined. Once the test condition is lifted, the GT–EP proceeds as though the intervening test instructions did not take place.

The GT–EP supports a large instruction and data space. The GT–EP supports 1GB of instruction memory and 256 MB of data memory. It has built–in circuitry to support data and instruction caches.

The GT–EP achieves 80–99% utilization of peak performance across a wide variety of applications as compared to 2–9% on conventional RISC and CISC architectures. A Whetstone benchmark performance of 32 million is achieved on the GT–EP when operating at 10 Mhz [Tan89].

3. Instruction Set

The GT–EP instructions are separated into four categories each of which can be specified independently. The four instruction categories are IAG, DAG, ALU, and I/O.

3.1. IAG

The IAG instructions are primarily used to control the generation of the program counter for fetching the next instruction from the instruction memory. The generation of the program counter is either direct or relative to a task pointer. In kernel mode, the generation of the program counter is direct. In user mode, the generation of the program counter is relative to a task pointer. The relative addressing scheme allows a user program to be dynamically relocated at run–time. Table 1 shows a summary of the IAG instructions. A detailed description of the IAG instructions can be found in [Tan90–1]. "Load" in Table 1 means loading a value from 1 of the 16 peripheral devices to the IAG module. "Store" in Table 1 means storing a value from the IAG module to 1 of the 16 peripheral devices.

Table 1. IAG Instruction Set Summary

Mneumonic	Description	Comments	Kernel Restriction
INV	Invalid Operation	Causes interrupt, useful for software trap	No
BALU condition branch address	Branch based on ALU status (zero, carray, sign, generic)	May specify branch prediction scheme and branch if true or false	No
BI branch address	Conditional branch based on external status and internal IAG status	No branch prediction is necessary	No
BP procedure address	Begin procedure	May specify to use software stack or hardware stack	No
BT task offset	Begin task	Begin execution at the user level	Yes
BL loop count	Begin loop	Setting up a loop between the BL and EL instructions	No
EL branch address	End loop	Branch if loop count is zero	No
EP	End procedure		No
CONT	Continue	Move to the next instruction	No
ET	End task		No
ECS	End context Switch	Return program control to the user	Yes
BX	Branch external	Branch address specified in the program counter register	No
BLX	Begin loop external	Loop count is specified in the loop counter register	No

ELX	End loop external	Loop branch address is specified in the program counter register	No
BPP	Begin privileged procedure	Used by the user to perform kernel service routine	No
EPP	End privileged procedure		No
EI	End Interrupt	Return from interrupt	Yes
BTX	Begin task external	Task offset is specified in the program counter register	Yes
HLT	Halt program execution		Yes
RST	Reset		Yes
ld_im	load interrupt mask	select a set of interrupts to monitor	Yes
ld_ndim	load non–disabled interrupt mask	select a set of interrupts that cannot be automatically disabled	Yes
ld_intr_v	load interrupt vectors	set the address of the interrupt service routines	Yes
en–intr	Enable interrupt	Enable the interrupt server	Yes
dis_intr	Disable interrupt	Disable the interrupt server	Yes
ld_tp	Load task pointer		Yes
ld_tpc_ul	Load task pointer upper limit	set the limit of the user instruction space	Yes
en_tp	Enable task pointer	Turn on the user mode	Yes
Dis_tp	Disable task pointer	Turn off the user	Yes
ld_ita_sv	Load interrupt timer A starting value		Yes
rst_ita	rest timer A		Yes
rst_itb	Reset timer B		Yes
ld_pc_reg	Load program counter register	The pc register is used by BPX, BTX, ELX, and Bx instructions	No
ps_pc_stk	Push pc stack	Push a value to the hardware stuck	Yes
en_stk_ovfli	Enable stack overflow indicator	Used to set the stack underflow flag when context switch	Yes
dis_stk_ovfli	Disable stack overflow indicator		Yes
ld_lcntr	Loan loop counter	The loop counter is used by the BLX instruction	Yes
ld_inst_reg 0	Load instruction register 0	Yes	
ld_inst_reg_1	Load instruction register 1		Yes
wr_inst_pha	Write phase A instruction	Instruction memory consists of Phase A bank bank and phase B bank	Yes
wr_inst_phb	Write phase B instruction		Yes

rd_inst_pha	Read the instruction from phase A memory bank to the instruction register	The memory location is designated int he pc register	Yes
rd_inst_phb	Read the instruction from phase B memory bank to the instruction register		Yes
ld_iag_flag	Load internal IAG flags to the status register		No
clr_iag_ef	Clear iag error flags	Clear iag error conditions	Yes
ld_io_t	Load io timer	io timer is used to monitor the freeze condition	Yes
rst_io_t	reset io timer		Yes
start_io_t	Start io timer		Yes
ld_bpp_reg	Load privileged procedure address	The privileged procedure address is used by the BPP instruction	Yes
set_bpp_stat	Set the privileged procedure status	It is useful when context switching	Yes
rst_bpp_stat	Reset the privileged procedure status		Yes
st_im	Store interrupt mask		No
st–ndim	Store non–disable interrupt mask		No
st_iv	Store interrupt vectors		No
st_tp	Store task pointer		No
st_tpc_ul	Store task pointer upper limit		No
st_ita	Store interrupt timer A		No
st_itb	Store interrupt timer B		No
st_ta	Store timer A	Store the current value of the timer	No
st_tb	Store timer B		No
st_pc_reg	Store pc register		No
pop_pc_stk	Store and pop pc stack		No
st_lentr	Store loop counter		No
st_inst_reg_0	Store instruction register 0		No
st_inst_reg_1	Store instruction register 1		No
st_iag_flag	Store IAG flag	IAG flag is stored in the status register	No
st_io_t_sv	Store io timer starting value		No
st_io_t	store io timer		No
st_bpp_reg	Store privileged procedure address		No

3.2. DAG

The DAG instructions are primarily used to control the generation of three address fields for fetching and storing data to and from the peripheral devices. The three address fields are denoted as *F_adr[25:0]*, *R_adr[25:0]*, and *S_adr[25:0]*. The *F_adr[25:0]* selects an address location for the output device designated by the *ods[3:0]* I/O instruction field. The *R_adr[25:0]* selects an address location for the input device designated by the *ids[3:0]* I/O instruction field. The *S_adr[25:0]* always selects an address location for the data memory, independent of the selection of *ids[3:0]*. The data memory is designated as *device 1*.

The generation of the address fields can be direct or relative. There are two levels of relative addressing. The first level of relative adressing is used to dynamically relocate user data space. The second level of relative addressing is used to allocate the run–time stack. The first level only applies to execution in user mode.

The following syntax summarizes the various addressing modes available for address calculations,

```
addr ::=   #off | #off[] | #off[]~ | %off | %off[] |
           %off[]~ | #^ | #^[]~ | %^ | %^[] | %^[]~
```

The # sign is used to indicate that the address is relative to a task pointer (used to dynamically reallocate user space). The % sign is used to indicate that the address is relative to an address pointer (used to access data stack). The ^ sign indicates that the address is calculated with a base offset stored in the base register. The *off* indicates that the base offset used is fetched directly from the instruction memory. The [] sign indicates that the address is calculated with an index offset stored in the index register. The ~ sign indicates that a post operation is to be performed on the index value at the end of the instruction execution. The post index operations are add or subtract and bit reversal.

Table 2 summarizes the instructions that control the registers that are used to generate the data addresses. "Load" in Table 2 means loading a value from 1 of the 16 peripheral devices to the DAG module. "Store" in Table 2 means storing a value from the DAG module to 1 of the 16 peripheral devices.

Table 2. DAG Load and Store Instructions

Mneumonic	Description	Comments	Kernel Restriction
lp_dtp	Load data task pointer	set the user data address offset	Yes
ld_tda_ul	Load task data address upper limit	set the user data address limit	Yes
ld_ap	Load address pointer	set the stack pointer	Yes
ld_apl	Load address pointer limit	Set the stack limit	Yes
incr_ap	Increment address pointer		No
decr_ap	Decrement address pointer		No
ld_idx	Load index register		No
ld_pidx	Load post index register	Specify the value to subtract or add for post index operation	No

ld_pidxm	Load post index mode register	Specify the post index operator	No
ld_base	Load base register		No
ld_inst	Load instruction register		Yes
ld_dag_flag	Load DAG status flags to the status register		No
clr_err_flag	Clear DAG error flags	Clear error conditions	Yes
ovr_odc	Override operand dependency check		No
st_dtp	Store data task pointer		No
st_dta_ul	Store data address upper limit		No
st_ap	Store address pointer		No
st_apl	Store address pointer limit		No
st_idx	Store index register		No
st_pidx	Store post index register		No
st_pidxm	Store post index register mode		No
st_base	Store base register		No
st_inst_reg	Store instruction register		No
st_dag_flag	Store DAG flags in the status register	Flags are loaded to the status register through ld_dag_flag instruction	No

3.3. ALU

The ALU supports three data types: real (R), integer (I), and bitfield (B). The real data type is represented by a 32–bit number in IEEE single precision format. The integer data type is represented by 24–bit sign magnitude numbers. The bitfield data type is represented by 32–bit unsigned numbers. Table 3 summarizes the ALU instructions that are supported by the GT–EP.

Table 3. ALU Instructions

Mueumonic	Data Types	Description	Comments
ADD	R,I	Addition	
SUB	R,I	Subtract	first operand minus second operand
RSVB	R,I	Reverse Subtract	second operand minus first operand
MULT	R,I	Multiply	
SHL	B	Shift left	
SHR	B	Shift right	
AND	B	Logical and	

OR	B	logical or	
XOR	B	logical xor	
NOT	B	logical not	bit inverse
PEXP	R	Pack exponent	Place operand in the real number exponent field
INVS	R	Inverse seed	Provide a first order division approximation
ROUND	I	Round to nearest	
PASS	R,I,B	Pass operand	No change is done on the operand
FLOAT	I	Convert integer to real	
UEXP	R	Unpack exponent	
UMAN	R	Unpack mantissa	Effectively zeros out the exponent field
REXP	R	Square root approximation for the exponent	
RMAN	R	Square root approximation for the mantissa	
SSIN	I,R	Sign of SINE function	Accepts an integer operand and a real operand
ODN	I,R	Odd negative	If the integer operand is odd, set the real operand to negative
SSWAP	I,R	Swap the sign of the two operands	
STAN	I,R	Sign of tan function	

3.4. I/O

The I/O category consists of two instruction fields: *ods[3:0]* and *ids[3:0]*. The *ods[3:0]* selects 1 of 16 output device for the output of the ALU. The *ids[3:0]* selects 1 of 16 input device for the input to the ALU.

4. Programming Examples

Two examples will be used to illustrate the steps involved in programming the GT–VIAG chip. The first example shows the steps involved in loading a program into the instruction memory. The second example illustrates running multiple tasks on the GT–VIAG chip. Both programs assume a reset condition as a starting point. On a reset, all interrupts are automatically disabled.

4.1. Program Loading

The GT–VIAG chip has a simple built–in loader. When the chip is reset, the GT–VIAG will begin program execution in kernel mode at instruction address 0. It will first look for a pattern of *0110* (MISC instruction) on the *pc_maj_op[3:0]* instruction field and a pattern of *xx–xxxx–xxxx–xxxx–0000–0101–1010* on the *pc_min_op[11:0]* instruction field. If the instruction patterns are found at instruction location 0, the GT–VIAG chip will proceed to fetch and execute the next instruction.

If the patterns do not exist (a most likely situation on a power up), the GT–VIAG will invoke the internal loader to begin fetching instructions from ids[3:0] channel 4. The internal loader program is as follows:

```
{ loading instruction memory }
0: ld_pc_reg(receive(4,#0));          { load loop counter}
1: ld_pc_reg(receive(4,#0));          { load starting address for instruction writes }
2: BLX                                { begin loop external }
3: ld_inst_reg(3,receive(4,#0));      { load instruction register 3 at GT–VDAG chip }
4: ld_inst_reg(2,receive(4,#0));      { load instruction register 2 at GT–VDAG chip for PHASE_A
                                        write }
5: ld_inst_reg(1,receive(4,#0));      { load instruction register 1 at GT–VIAG chip }
6: ld_inst_reg(0,receive(4,#0));      { load instruction register 0 at GT–VIAG chip }
7: ld_inst_reg(2,receive(4,#0));      { load instruction register 2 at GT–VDAG chip for PHASE_B
                                        write }
8: wr_inst_pha;                       { write the data in instruction registers 1, 2, 3, and 4 to the in-
                                        struction memory that is accessed on PHASE_A at the loca-
                                        tion specified by the program counter register }
9: wr_inst_phb;                       { write the data in instruction register 2 to the instruction
                                        memory that is accessed on PHASE_B at the location speci-
                                        fied by the program counter register and increment the con-
                                        tents of the program counter register by one }
a: EL(3)                              { check the contents of the loop counter; If the contents of the
                                        loop counter are not one, decrement the loop counter by 1
                                        and branch to location 3; load instruction register 3 at GT–
                                        VDAG chip }
{ loading data memory }
b: BALU(Z,F,e,receive(4,#0)) || ld_base(F,receive(4,#0))
                                      { load F base register and branch to address e if the received
                                        value is zero and assume branch is false }
c: CONT;                              { waiting for ld_base instruction to take effect }
d: BI(b) || #^ := receive(4,#0);      { assign the received value to the data memory location speci-
                                        fied by the F base register and branch to address b }
```

e: #0 := receive(4,#0); { assign the received value to data memory location 0 }

f: rst; { reset the processor }

With the exception of the wr_inst and rd_inst instructions, all of the GT–VIAG instructions take 1 cycle to execute. The wr_inst and rd_inst instructions take two cycles to execute. The body of the loop for loading an instruction field consists of 5 load instructions, 1 nop (continue), 2 write instructions, and 1 end of loop instruction for a total of 9 cycles. If a 100–ns cycle time is used (the speed of an existing prototype), it will take 58.98 ms to load 64k of instruction words. On the other hand, if a 300–ns data transmission cycle time is assumed on channel 2, it will take 98.304 ms to load 64k of instruction words because 5 transfers are required per instruction word.

4.2. Vector Multiply

An axample will be used to illustrate the various addressing modes of the GT–VDAG chip. A hand translation of a simple Pascal program will be presented.

The Pascal program to be considered is as follows:

```
Program Test;
const
  max_dim = 100;
type
  vector : array[1..max_dim] of real;
var
  i,N : integer;
  x,y,z : vector;

procedure vector_multiply(var c:vector;a,b:vector;dimension:integer);
var i : integer;
  procedure display_vector;
  var i : integer;
  begin
    writeln(i);
    writeln(a[i]);
    writeln(b[i]);
    writeln(c[i]);
  end; { of display_vector }
begin
  for i := 1 to dimension do
    c[i] := a[i] + b[i];
  for i := 1 to dimension do
    display_vector;
end; { of vector_multiply }

begin
  readln(N);
  if N <= max_dim then
```

```
  begin
    for i := 1 to N do
    begin
      readln(x[i]);
      readln(y[i]);
    end;
    vector_multiply(z,x,y,N);
  end;
end.
```

The global variables are assigned with absolute addressing mode as shown in Table 4.

Table 4. Global Variable Assignment

Variable/ Constant	Absolute Location #
Max_dim	0
i	1
N	2
x	3..102
y	103..202
z	203..302
1	303
203	304
202	305
101	306
vector_multiply_ap	307

The variable *vector_multiply_ap* is used to store the address pointer of the procedure *vector_multiply*. It is needed because procedure *vector_multiply* contains a nested procedure, *display_vector*, and the nested procedure requires access to the local variables of procedure *display_vector*. The integer constant 203 is used to pass the starting address of array *z* as a *call–by–reference* parameter. Integer constants 202 and 101 is used to access the local array *a* and *b* of procedure *vector_multiply* from the procedure *display_vector*.

The local variables of procedure *vector_multiply* are assigned with relative addressing mode as indicated in Table 5.

Table 5. Local Variable Assignment of Procedure Vector_multiply

Variable/ Constant	Relative Location %
c^	0
a	1..100
b	101..200
dimension	201
i	202
temporary	203

The variable *c^* is used to store the pointer to the starting address of the *call–by–reference* array *c*.

The main body of the program is translated to the GT–VIC instructions as follows:

```
{ readln(N) }
0: #2 := receive(3);          { receive N from channel 3; readln statement is translated as
                                a receive instruction from channel 3 }
{ if N <= max_dim then }
1: BALU(S,F,19,#0–#2);        { subtract #2 from #0, branch to location 19 if the sign flag is
                                1, assume branch is not taken }
{ for i := 1 to N do }
2: %0 := #2 – #0;             { set temporary variable at relative location 0 and assigned it
                                to the loop count }
3: ld_pc_reg(%0);             { load the program counter register with the loop count }
4: ld_idx(f,0);               { load the f index register with a constant 0 }
5: ld_pidxm(frs,+);           { set the f, r, and s post index operators to perform addition }
6: ld_pidx(frs,1);            { load the f, r, and s post index registers with a constant 1 }
7: BLX;                       { begin loop with a count value in the program counter register
                                }
{ readln(x[i]) }
8: #3[] := receive(3);        { receive x[i] from channel 3 by using a constant address offset
                                of 3 and the indexing mode }
{ readln(y[i] }
```

9: EL(8) || #103[]~ := receive(3); { receive y[i] from channel 3 by using a constant address offset of 103 and the indexing mode. The ~ operator is used to activate the post index operation which has been setup to add 1 to the content of the index register. The EL(8) instruction will branch to instruction 8 and decrement the loop counter by 1 if the loop counter is not a 1 }

{ vector_multiply(z,x,y,N) }

10: ld_pc_reg(%1); { load program counter register with loop count of N–1 }

11: ld_idx(fr,0) { load f and r index registers with a constant 0 }

12: incr_ap(204); { increment the address pointer by 204, which is the amount of local and temporary data space required to execute procedure vector_multiply }

13: %0 := #203 { pass the starting address of array z to local variable c^ }

14: BLX; { begin loop with the loop counter from the program counter register }

15: %1[] := #3[]; { copy array x into local array a of procedure vector_multiply }

16: EL(15) || %101[]~ := #103[]~; { copy array x into local array b of procedure vector_multiply; post increment the f and r index registers; If the content of the loop counter is 1, goto 15 and decrement the loop counter by 1 }

17: BP(19) || %201 := #2; { passing the value of N to the parameter *dimension*; begin procedure vector_multiply at location 19 }

{ end. }

18: ET; { end of task and return program control to the kernel }

The procedure *vector_multiply* is translated into the GT–VIC instructions as follows:

19: #307 := st_ap; { store the address pointer of procedure vector_multiply. This is necessary because procedure vector_multiply contains a nested procedure and the nested procedure requires access to the local variables of procedure vector_multiply }

{ for i := 1 to dimension do }

20: %203 := %201 – #303; { temporary variable := dimension – 1 }

21: ld_pc_reg(%203); { load program counter register with the loop count }

```
22: ld_idx_reg(frs,0);          { load f, r, and s index registers with a contant 0 }
{ c[i] := a[i] + b[i] }
23: ld_base(f,%0);              { load f base register with the starting address of array c }
24: BLX;                        { begin loop with a count value stored in the program counter
                                  register }
25: EL[25] || #^[]~ := %1[]~ + %101[]~;
                                { c[i] := a[i] + b[i]; repeat until the content of loop counter
                                  is 1 }
{ for i := 1 to dimension do }
26: %202 := #303;               { i := 1 }
27: BALU(NS,F,29,%201–%202);    { if dimension > i then goto 27; assume branch is not taken }
{ display_vector }
27: BP(32) || incr_ap(1);       { begin procedure; increment the address pointer by 1 to allow
                                  1 temporary variable for procedure display_vector }
{ end; { of for } }
28: BI(27) || %202 := %202 + #303;
                                { i := i–1 }
{ end; { of procedure vector_multiply } }
30: decr_ap(204);               { decrement address pointer by 204 to release the local and
                                  temporary work space of procedure vector_multiply }
31: EP || #307 := st_ap;        { end of procedure and restore the address pointer of procedure
                                  vector_multiply; The later is needed only if a procedure is re-
                                  cursively called. }
```

The procedure *display_vector* is translated into the GT–VIC instruction as follows:

```
{ writeln(i) }
31: %1 := #307 + #305;          { calculate the address for accessing i : address pointer of vec-
                                  tor_multiply + constant offset of variable i }
32: ld_base(r,%1);              { load r base register }
33: ld_idx(r,%1);               { load index register with the calculated address of i for later
                                  use }
34: send(3,^);                  { send i to channel 3 using indirect addressing mode }
```

```
{ writeln(a[i]) }
35: %1 := #307 + #306;          { calculate base address of vector a }
36: ld_base(r,%1);              { load r base register }
37: cont;                       { wait for load base instruction to take effect }
38: send(3,^[]);                { send a[i] }
{ writeln(b[i]) }
39: %1 := #307 + #303;          { calculate base address of vector b }
40: ld_base(r,%1);              { load r base register }
41: cont;                       { wait for load base instruction to take effect }
42: send(3,^[]);                { send b[i] }
{ writeln(c[i]) }
43: ld_base(r,#307);            { load c^ base address; note c^ is at relative address 0 }
44: cont;                       { wait for load base instruction to take effect }
45: send(3,^[]);                { send c[i] }
{ end { of display_vector } }
46: EP || decr_ap(1);           { end of procedure and decrement address pointer by 1 }
```

The kernel that invokes the execution of the program is as follows:

```
0: ld_tp(1000);                 { load task pointer; allocate instruction execution beginning at
                                  location 1000 }
1: ld_tpl(1046);                { load task pointer limit; the last valid instruction address of
                                  program test is at location 46 }
2: ld_dtp(2000);                { load data task pointer; allocate data space for program test
                                  beginning at location 2000 }
3: ld_apl(2308);                { load address pointer limit; the last global variable is at loca-
                                  tion 207 }
4: ld_ap(2607);                 { load address pointer; provide 300 locations for data stack }
5: ld_tda_ul(2067);             { load task data address upper limit; allow 1 temporary vari-
                                  able for the main program }
```

– load program into the instruction memory

– load constants into the data memory

n+1: BT(0); { begin execution of program *test* at relative instruction address 0 }

n+2: hlt; { halt the processor, mission accomplished }

The above example illustrates the use of the various addressing modes on the GT–VDAG module.

4.3. Multi–tasking

The following example illustrates the creation of two tasks and the setting up of an interrupt service routine for context switching between two tasks. When the two tasks are completed, the processor is halted.

The following constants are assumed to be available from the data memory :

tp_a : task A task pointer;
tp_b : task B task pointer;
tb_adr : timer B interrupt address;
cs_adr : context switch instruction address;
tb_sv : timer b starting value;
tb_im = 00008000 : constant for enabling timer B;
zero = 00000000.

The following constants are used as a part of the instruction field :

A_adr : task A instruction address;
B_adr : task B instruction address.

{ The following instructions are executed when the processor is reset } 0:

CONT; { first instruction executed after a reset }

– setup task table

ld_intr_v(15,tb_adr); { setup the interrupt address for timer B }

ld_itb_sv(tb_sv); { load timer B starting value }

ld_im(tb_im); { enable timer B for context switch }

ld_tpa(tp_a); { setup the task pointer for task A }

rst_itb; { reset timer B and turn on the timer }

– mark that task A is completed has been started and was last executed

BT(A_adr); { begin task A and turn on the task pointer mode (user mode) }

– mark that task A has been started and was last executed

```
BI(cs_adr);                    { branch to the context switch routine }
```

{ The following instructions are executed when timer B generates an interrupt (when a carry out is generated). }

```
tb_adr:

CONT;                          { always needed as the first instruction in a service routine }

dis_tp;                        { disable the task pointer mode/user mode }

push_pc(cs_adr);               { push context switch address onto the program counter stack;
                                 forcing the return interrupt address }

end_intr                       { end interrupt }
```

{ The following instructions are executed on a return from timer B interrupt }

```
cs_adr:

– check the information on the task table

case 1: task A is active and task B is active

        case 1.1: task A was last executed and task B has not been started

– save task A processing state

BT(B_adr)                      { begin task B at location B_adr; turn on the task pointer
                                 mode } – mark task B as complete

BI(cs_adr)                     { branch to context switch handling routine }

        case 1.2: task A was last executed and task B has been started

– save task A processing state

– restore task B processing state

ECS;                           { end of context switch; turn on the user mode and begin execution as though task B has just been returned from the timer interrupt }

        case 1.3: task B was last executed and task A is active

– save task B processing state

– restore task A processing state

ECS;                           { end of context switch; return to task A }

case 2: task A is complete and task B is active
```

```
        case 2.1: task A was last executed
    – restore task B processing state
    ECS;                        { end of context switch; return to task B }
        case 2.2: task B was last executed
    ECS;                        { end of context switch; return to task B }
    case 3: task A is active and task B is complete
        case 3.1: task B was last executed
    – restore task A processing state
    ECS;                        { end of context switch; return to task A }
        case 3.2: task A was last executed
    ECS;                        { end of context switch; return to task A }
    case 4: task A and B are complete.
    hlt;                        { halt the processor , mission accomplished }

{ The following instruction is executed when BT(A_adr) is executed } A_adr:
    – task A program body
    ET;                         { return control to the kernel section that creates Task A }

{ The following instruction is executed when BT(B_adr) is executed }
     B_adr:
    – task B program body
    ET;                         { return control to the kernel section that creates Task B }
```

The program illustrated above provides a brief overview of how to program the GT–VIAG chip to perform multi–tasking. The key instructions for the context switch are BT, dis_tp, push_pc, ECS, and ET. The BT instruction is used create a task to transfer control from the kernel mode to the user mode. The dis_tp instruction is used to explicitly set the kernel execution mode in the timer interrupt handling routine. The push_pc instruction is used to force the context switch address as the interrupt return address. The ECS instruction is used to return program control from the kernel to the user mode as though it is returned from the timer interrupt service routine. Finally, the ET instruction is used by the user to complete the execution of the task.

5. References

[Back78] Backus, J., "Can Programming be Liberated from the von Neumann Style ? A Functional Style and its Algebra of Programs," Communications of the ACM, Vol. 21, No. 8, pp. 613–41.

[Tan89] Tan, W. S., *A VLSI Parallel Processor Structure for Scientific Computing*, Ph.D. dissertation, Georgia Institute of Technology, May 1989.

[Myer78] Myers J. G., *Advances in Computer Architecture*, New York, N.Y.: John Wiley & Sons, 1978.

[Patt82] Patterson D. A. and Sequin C. H., "A VLSI RISC," Computer, Sept. 1982, pp. 8–20.

[Gupt83] Gupta, A. and Toong, H. D., "An Architectural Comparison of 32–bit Microprocessors," IEEE Micro, Feb. 1983, pp. 9–22.

[Tan90–1] Tan, W. S. and S. H. Russ, Instruction Address Generation: A Programming Model, CERL008–0061.1, Georgia Tech, June 1990.

[Tan90–2] Tan, W. S. and S. H. Russ, Data Address Generation: A Programming Model, CERL008–0062.1, Georgia Tech, June 1990.